Published in the 1995 IEEE Conference on Robotics and Automation

The ControlShell Component-Based Real-Time Programming System

Stanley A. Schneider
Real-Time Innovations, Inc.
954 Aster
Sunnyvale, California 94086

Vincent W. Chen
Gerardo Pardo-Castellote
Aerospace Robotics Laboratory
Stanford University
Stanford, California 94305

Abstract

Real-time system software is notoriously hard to share and reuse. This paper walks through the methodology and application of ControlShell, a component-based programming system real-time system software development. ControlShell combines graphical system-building tools, an execution-time configuration manager, a real-time matrix package, and an object name service into an integrated development environment. It targets complex systems that require on-line reconfiguration and strategic control.

ControlShell takes advantage of functional object hierarchies to enable code sharing and reuse. It gains flexibility by supporting easy interconnectivity of these objects. It features a unique configuration control system for changing operating modes.

The paper concludes by examining the application of this framework to a dual-arm robotic workcell that is able to pick objects from a moving conveyor and perform simple assemblies.

1 Introduction

Motivation System programs for real-time command and control are, for the most part, custom software. Modern real-time operating systems [1,2,3,4] provide some basic building blocks—scheduling, communication, etc.—but do not encourage or enable any structure on the application software. Information binding and flow control, event responses, sampled-data interfaces, network connectivity, user interfaces, etc. are all left to the programmer. As a result, each real-time system rapidly becomes a custom software implementation. With so many unique interfaces, even simple modules cannot be shared or reused.

An effective real-time programming environment must facilitate sharing and reuse of program modules. It must assist the programmer both in structuring complex systems and in managing the system at run time. The framework must also provide services and tools to combine modules and build systems from reusable components. Finally, it must meet the many challenges unique to real-time computing, such as reacting to external temporal events, blending strategic-level command and low-level servo control, and switching between different modes of operation.

Summary This paper presents the rationale and motivation behind the *ControlShell* component-based programming system. ControlShell implements several new design concepts that have been proven effective in hard practice. For instance, while it is object-oriented, ControlShell differs from other object-oriented systems in that ControlShell objects implement functional units rather than model physical system components. ControlShell takes advantage of inheritance of these objects to provide complex functionality that is easily shared and reused. Also, these objects retain their identity in the run-time system, and are entered into an object name service. This allows object to cross-reference each other, facilitatiing integration.

ControlShell also addresses the fundamental issue of how to best merge event-driven reaction with feedback control. ControlShell presents a unique approach that uses an event-reaction programming system to change the data-flow pattern in a block-diagram model—without necessarily changing the diagram itself. This approach makes small and large changes in code configurations equally easy to implement. The result is a simple model that encourages fine mode control, and thus fine state definition, while providing considerable flexibility and generality.

2 Approach

To our knowledge, ControlShell is the only framework combining component-based data-flow system construction, event-driven state programming, a run-time executive, and transparent network connectivity.

ControlShell was written to address a specific domain: complex electro-mechanical systems. It has been driven by practical applications from its inception[12,13,14]. We start by describing the problem domain and objectives. We then examine ControlShell's architecture in the perspective of the many other approaches to real-time software development.

2.1 Strategic Objectives and Architectural Decisions

Practical Focus ControlShell is driven by practical, immediate considerations. It intentionally does not address design-time verification of real-time deadline constraints, nor guarantee that state machines will not deadlock. Instead, ControlShell assumes its target systems will be easily subjected to run-time validation both in simulation and on real hardware. Many tools are provided for addressing these issues at run time (execution profiling, flexible missed-deadline response, etc.). This trade-off has proven, in practice, to be very effective. For instance, it frees developers from making difficult estimates of execution times of complex code, and allows a more general state-programming model and complex state-transition action routines. We have chosen, essentially, practical flexibility over provable correctness, at the (minor) cost of extra run-time testing.

Reuse and Sharing ControlShell concentrates on code reusability and sharing. This is one of the factors that drove the development of the "functional" object hierarchies. By building functional units rather than modeling physical system components, ControlShell developers can take advantage of complex system building blocks that can be applied to many applications. We thus strike a compromise between functional block-diagram tools (see below), and the power of object-behavior inheritance.

Programming System. ControlShell is designed, from the start, as a programming system. While many systems can be built without custom code by linking pre-existing libraries, the emphasis has always been on providing a development environment that talented programmers will be happy to work with. ControlShell strives always to allow creative users to implement inventive solutions beyond the framework designer's original intent.

Separable services ControlShell is structured as a set of inter-related "services". Rather than providing an ultimate integrated solution, ControlShell provides tools that make sense in a complex system. Integration is accomplished by providing extensive interface modules. However, all interfaces between services are open; users may choose to replace most any portion of the system with designs (or research results) more to their own liking. This decision has resulted in a flexible system that still works together (nearly) seamlessly.

Interconnectivity Complex systems often have inter-module interactions. For instance, an event-driven strategic control module must be able to interact with motion controllers, low-level routines must be able to interact with each other and raise conditions that higher levels handle. In a complex (or research) system, these interactions are often difficult to foresee or even characterize. ControlShell addresses this challenge by a) retaining the identity of all design-time objects in the run-time system, and b) providing a run-time "object name service", so any module may look-up any object at run time. This design provides very flexible connectivity.

Networking ControlShell is integrated with a network connectivity package called the Network Data Delivery Service (NDDS). NDDS is a novel network-transparent data-sharing system. It implements a "subscription" data-passing model that allows multiple clients to transparently and anonymously communicate data on a network. However, this functionality is beyond the scope of this paper, see [5] for details.

2.2 Perspective

There are many approaches to developing real-time system software, far too many to analyze them all here. Instead, we attempt to survey the general categories of tools, and differentiate their approaches from ControlShell's.

Hierarchy Specifications There are two quite different issues in real-time software system design: hierarchy (what is communicated), and superstructure (how it is communicated).

Several efforts are underway to define hierarchy specifications; NASREM[6] and UTAP[8] are notable examples. ControlShell makes no attempt to define hierarchical interfaces, but rather strives to provide a sufficiently generic software platform to allow the exploration of these issues.

Block Diagram Editors There are several functional block diagram editors and code generators. These include SystemBuild/AC100 by Integrated Systems Inc., and SIMULINK (a.k.a. the Real-Time Workshop) by The MathWorks, Inc.[11,3] These tools are heavily biased toward the low-end controls market. As such, they have interfaces to control design tools, and are powerful for choosing gains, designing controllers, etc. However, while they do have some facility for "custom" blocks, they are not designed as programming systems. Code generated from the block diagram combines both data objects and functional blocks into monolithic structures at run time. As a result, there's little interconnectivity and limited ability to develop complex, custom systems. These tools also do not address event-driven reactive programming.

Real-Time Formalism Tools There are several "traditional" real-time formalisms. Products exist on the market that implement some of these. For instance StateMate[7] by iLogix is based on Karel's StateCharts, ObjectTime is

based on a Ward-Mellor variant called ROOM[19].

These systems concentrate on state machine behavior and interaction. They target a different application audience than ControlShell; they are mostly aimed at systems that are complex due to concurrency, such as a telecommunications system connected to 400 lines.

ObjectTime (ROOM) is object oriented. However, ROOM defines objects by modelling physical system components as state machines. A state machine in ROOM is a single object. This is fairly large-grained view of the world. In ControlShell, the state-transition routines are the objects. That allows users to develop reusable libraries of transition routine objects, and build on base classes to construct complex action routines. This is advantageous in systems that perform non-trivial processing in the action routines. ObjectTime does not deal with data-flow or feedback configuration issues.

Onika [20] is a "software composition system" from CMU. On the surface, it resembles ControlShell in that it composes systems from blocks of code. However, there are many major differences. Blocks in Onika are independent tasks. This forces a very "large-grained"' model, and is subject to loop delays. Onika supports simple changes in configurations (by substituting an entirely new diagram), but not graphical definition of complex configurations or event-driven reconfiguration. Onika does not address object-oriented issues, state programming, network connectivity, or automatic code generation.

ORCAD/ESTEREL The ORCAD system[10] provides an object-oriented design approach for robotic systems. ORCAD combines control laws and reactive behaviors into objects called ROBOT-TASKS. Along with the ESTEREL language, ORCAD strives for formal verification of temporal properties of control programs.

As in Onika, blocks (called MODULE-TASKS) in ORCAD each run in a separate task context, and are thus large-grained. While state automata are provided, there is no notion of state programming, nor active transition routines. Operating mode switching is supported only by redefining the entire block diagram for each ROBOT-TASK.

The next section analyzes ControlShell's system design methodology in the context of the issues presented above.

3 Methodology

3.1 Data-Flow Design Methodology

We term any system that has a periodic execution cycle a *data-flow system*. This includes most control loops, data acquisition systems, etc. These systems are sometimes also referred to as *sampled-data systems*.

Design Cycle The data-flow design process is shown in Figure 1 To design a data-flow system, the developer must first break the system into manageable components. Then each component is either implemented or selected from a library. The next step is to connect the components into an operational system. Testing the overall system provides the feedback required to drive a successful design.

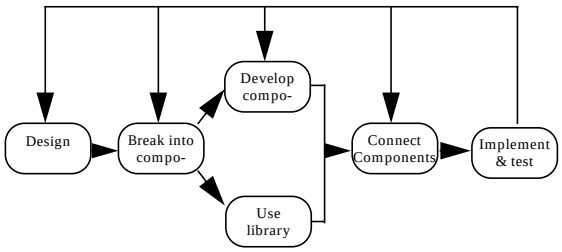


Figure 1. Data-Flow Design Cycle

ControlShell provides tools for every phase of the data-flow design cycle.

Components ControlShell builds data-flow systems from small, reusable objects called *components*. Components implement a specific functionality within a sampled-data environment via methods that run at well-defined times, such as at each sample-clock tick. By allowing components to attach easily to these critical times in the system, ControlShell defines an interface sufficient for installing (and therefore sharing) generic sampled-data programs.

Components read input signals, generate output signals, and use reference signals. Signals many be any of several types; most are named matrices called *CSMats*. Reference signals are often used for parameters, such as gains, names of other objects, or file names from which to load data. Figure 2 shows an example of the *pdControl* component that implements a simple Proportional-Derivative controller.

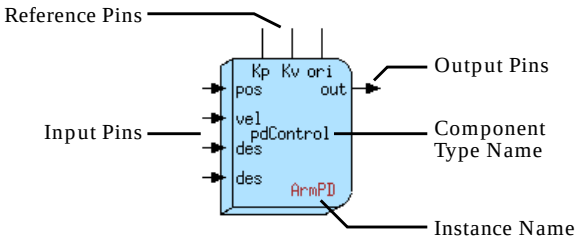


Figure 2. Example Component

Each component is labeled with a *type name* and an *instance name*. The instance names allow components to

be reused in the same diagram. For example, the *pdControl* component's instance name in Figure 2 is *ArmPD*. Instance names are registered with *ControlShell*'s object name service. That way, other components and other *ControlShell* facilities—such as the finite-state machine—can bind to them at run-time and call their public external methods, etc. This easy connectivity is a critical feature that allows *ControlShell* to support arbitrarily inter-related diagrams.

Each type of component is implemented as a C++ class. Components are derived either from a common base class, or from other components. Thus, components are built into class hierarchies of similar functionality. Derived components may hide (e.g. default) functionality or parameters to form an easier-to-use component, or add functionality to the base class, forming a more complex or functional component.

An example of a more functional derived class is a component called *CmdNddsConsumer*. The CmdNddsConsumer component is a derived class of a base class that interfaces to NDDS. The base class subscribes to network data items and provides them for use by the data-flow system. The derived class also gets the data from the network, but then sends a stimulus to a state machine announcing the arrival (See Figure 3). This provides a powerful and simple means of implementing network-distributed data flow and reactive behavior.

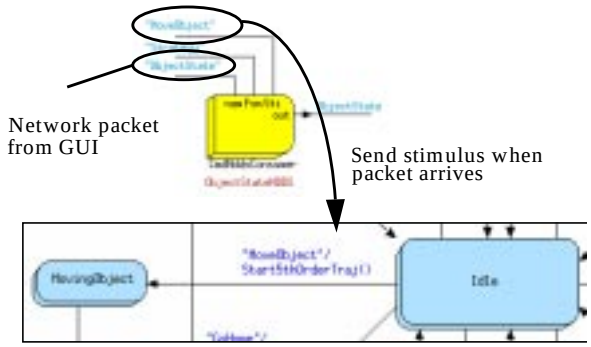


Figure 3. Functional Class Hierarchies

Sophisticated actions can be built from simpler functional classes. Here, the arrival of a network command causes a dual arm robotic system to move an object.

A library of pre-defined components is provided, ranging from hardware device drivers and controllers to trajectory generators and sophisticated motion planning modules. New or custom components are easily added to the system via a graphical data interchange editor and C++ code generator. This tool makes building and maintaining hierarchies of components simple to manage.

Connections Components are connected within a graphical tool called the *Data-Flow Editor* (DFE), shown in Figure 5. A system may be built from many separate block diagrams.

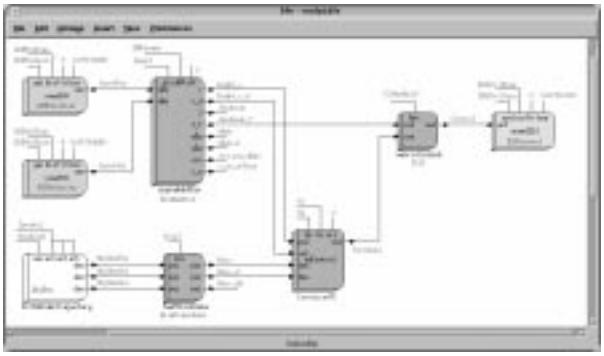


Figure 4. The Data-Flow Editor

The Data-Flow Editor connects components into systems. Blocks are components; lines on the diagram represent matrix objects. This diagram is a Cartesian-space controller for a 4-DOF SCARA robot.

Multiple diagrams are coordinated via the system manager. The system manager builds executing systems from sets of DFE diagrams. In Figure 5, two systems are set up. The system named "vx" will execute the actual hardware, the system named "unix" will execute the simulation. This capability makes hardware-in-the-loop simulation simple to set up. A similar setup lets the same high-level code run on different hardware configurations.

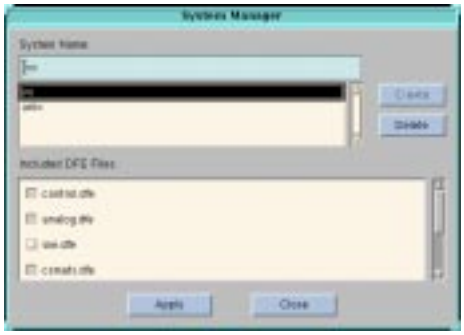


Figure 5. The DFE System Manager

The System Manager makes it easy to easily mix and match subsystems. In this case, it is being used to alternate between live execution and simulation.

Execution The DFE outputs a textual language that describes all the connections in the system. The run-time executive parses the system description file, loads the required components, and dynamically links the signals specified in the block diagram (using the object service). New diagrams may be loaded at any time. Thus, ControlShell systems can be dynamically updated.

All component objects are placed on dynamic lists. The run-time executive orders the lists, thus scheduling the components' execution order to minimize delay. All components (that execute at the same sample rate) may then run as a single task (execution context). In multi-rate designs, a separate task is used to execute each sample rate present in the system. This design maintains each object's identity, while eliminating task-switch overhead between blocks.

Configurations Complex real-time systems often have to operate under many different conditions. The changing sets of conditions may require drastic changes in execution patterns. For example, a robotic system coming into contact with a hard surface may have to switch in a force control algorithm, along with its attendant sensor set, estimators, trajectory control routines, etc.

ControlShell's configuration manager directly supports this type of radical behavior change; it allows entire groups of modules to be quickly exchanged. Thus, different system personalities can be easily interchanged during execution. This is a great boon during development, when an application programmer may wish, for example, to quickly compare controllers. It is also of great utility in producing a multi-mode system design. By activating these changes from the state-machine facility (see below), the system is able to handle easily external events that cause major changes in system behavior.

Configurations are defined at design time by assigning components to groupings called *module groups* and *categories*, see [15] for details. System mode changes are then effected by the run-time configuration manager. The manager quickly reconfigures large numbers of active component objects, essentially redirecting the data-flow paths through the diagram.

This design offers much finer configuration control than other systems. By allowing the designer to implement many configurations on a single diagram, it eliminates the problems with maintaining several similar diagrams. It also encourages small changes in data-flow where appropriate. The named configurations are (of course) C++ objects, and are listed with the object service. Thus, they may be bound at run time by any module and activated when needed. The state programming system (discussed next) makes good use of this feature.

3.2 State Programming Design Methodology

All complex systems must be strategically guided. Since real-time systems must operate in a complex, event-rich environment, this means that the strategic control must react to many events. However, sequential processing is not well-suited to managing events. Event-driven programming—defining a sequence of events and the actions to take when the events occur—is much more appropriate. We term this *state programming*, because the process consists of identifying system states and the events recognized in those states.

Design Cycle The strategic control design process is shown in Figure 6. To design a strategic control system, the developer must first formalize the situation—identify the possible events the system may encounter, and specify what action the system should take in response to those events. The next step is to implement the action routines, or select them from a library. The final step is to connect the events to actions. Implementing and testing the design provides the feedback that makes the system work.

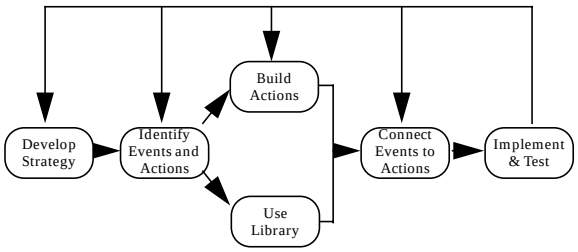


Figure 6. Strategic Control Design Cycle

ControlShell formalizes and assists reactive strategic programming.

To support strategic programming, *ControlShell* provides a state-machine programming system, consisting of a real-time state-machine engine, a graphical state-machine editor, and a state-transition-module (action routine) generation and management system.

Transition Modules The *ControlShell* state-programming system uses executable objects that implement actions in response to events. Because actions often result in state transitions, the objects are called *transition modules*. Transition modules implement a specific action, but are not intrinsically bound to an event. As with components, complex actions can be built by creating class hierarchies of transition modules.

Transition modules can accept parameters. For instance, a transition routine that activates a trajectory generator may take as parameters the name of the trajectory generator, the goal position, the slew time, and even the name of a configuration to activate before starting the trajectory. At run time, this transition routine will use the object data service to "hook-up" with the appropriate trajectory generators, configurations and data.

New transition modules are created via a graphical editor that defines the module's name, base class, formal parameter list, and possible return codes. A C++ code generator

generates the code required to interface the new object to the system.

The ability to accept parameters combined with the ability to inherit the functions of existing transition modules makes transition modules easy to share and reuse.

Connections Transition modules are bound to events within the graphical State Programming Editor (SPE). Events in ControlShell are defined as boolean expressions of stimuli, where stimuli can be assertions (e.g. “Power = on”) or triggers (e.g. “Contact”). Specifying a state transition therefore requires specifying a) a boolean expression (rule) that triggers the transition, b) the action (transition module) to execute, and c) the possible next states, depending on the return status of the transition module. All these are entered within the graphical SPE tool shown in Figure 5. The result is a graphical description of the

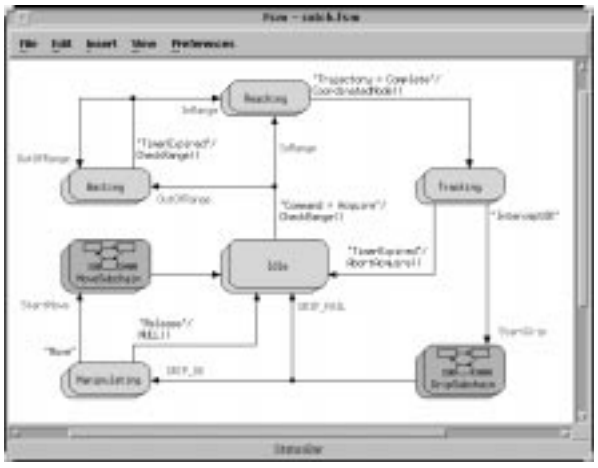


Figure 7. The State Programming Editor

Blocks in the SPE are states, the arrows represent transitions. The arrow labels are the boolean transition trigger expressions and transition module names. Transition parameters are set by clicking on the labels.

events and actions required to complete a task: a state program.

In addition, the tool allows assigning values to the transition module’s parameters. As with components, transition modules may use the object service to bind to any other object in the system. For example, a transition module that wants to take the action of moving a robot arm could look up and activate the control configuration that will drive the arm, and then find the trajectory generator that will cause the motion and start it. If this action is tied to the stimulus generated by the CmdNddsConsumer discussed above, it will allow a complex motion to occur in response to a network command. This easy integration results in considerable power.

State Machine Engine The real-time state machine engine is designed to provide strategic control, while also managing concurrency in the system. ControlShell’s state machine model features rule-based transition conditions, true callable subroutine hierarchies, task synchronization and event management. The details are beyond this paper, see [15]. It is, however, worth noting that the callable state subroutine concept also encourages reusability of state programs.

Execution As with DFE files, the SPE generates a textual language description of the state program. This description is parsed and linked by the run-time executive, and can be updated dynamically at run time. Each state program is executed by a separate task; stimuli are sent through message queues to the task.

4 Application Example

Figure 8 is a picture of a “factory of the future” robotic workcell being developed at Stanford University[16]. It combines two cooperating robotic arms, a real-time vision system, a networked motion planner, a conveyor belt, and an interactive graphical user interface. The arms accept commands from the user, pick up parts from the conveyor, maneuver them around obstacles, and perform simple assemblies under vision-guided feedback. It is a complex, multifaceted, distributed control problem [18].

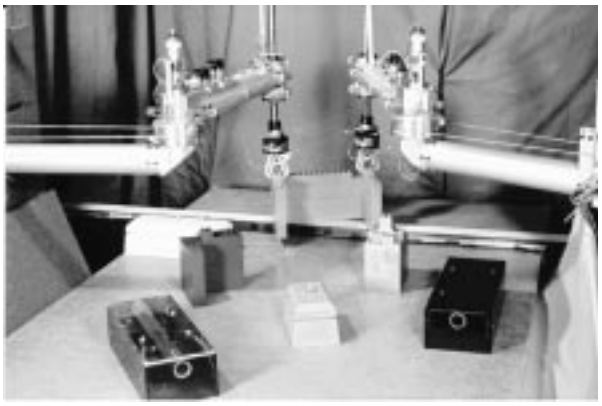


Figure 8. Cooperating Arm Workcell

We will concentrate on the problem of intercepting a part on the conveyor. The intercept strategy has three stages. In stage one, the arms are under control of an intelligent motion planner. The planned motion gets the arms close to the conveyor without hitting each other or any obstacles. However, the planner cannot plan motions fast enough to catch a moving object. In stage two, the arms change over to a local smooth-trajectory generator. The trajectory generator blends a smooth path from the arm's state to inter-

cept the object. The arms follow this path until they get "close enough" to the object. Then, they complete the acquisition by grasping the object, and switch to direct cooperative control of the object's position as measured by the vision system.

The system runs on five real-time VxWorks processors and two workstations. We will examine only the trajectory-generation processor's data-flow diagram. At each step of the operation, the control system needs to switch trajectory-generation strategies (and control modes).

The figures on the next page show some of the configurations required to generate these trajectories. In all cases, the goal of the diagram is to feed desired state outputs to the two shared-memory output components in the upper right corner. They feed into the control processor system. The components on the left are sensory inputs: conveyor inputs, vision system object estimates, force sensor readings. There are two types of trajectory generator components: the "via-point" generator follows a planner geometric path; "fifth-order" generators follow smooth local trajectories. The large component in the center implements an object impedance controller[13] for cooperative-arm manipulation.

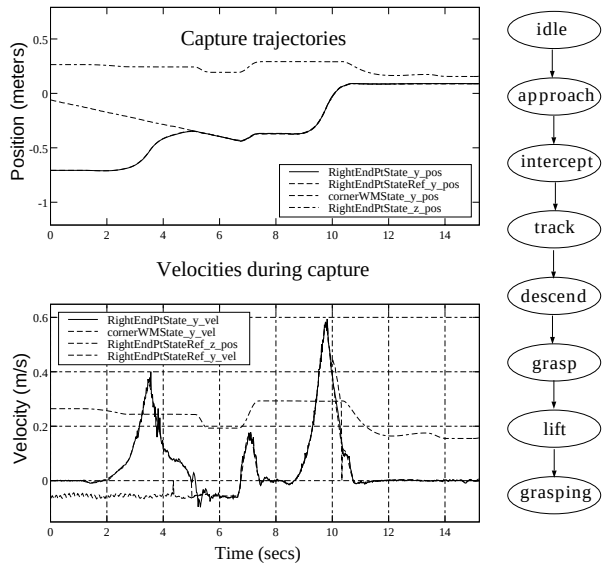


Figure 9. Capture Experiment

The arms switch from independent planned motion to local intercept trajectories at t=4 secs, and to tracking at t=5 secs. The capture is complete by t=7 secs. Note that both the positions and velocities match during the tracking and grasping phases.

Figure 9 shows a trace of one arm's motion, and a simplified version of the state program that switches configurations.

5 Conclusions

This paper has presented a brief overview of the philosophies behind the ControlShell system. ControlShell is designed—first and foremost—to be an environment that enables the development of complex real-time systems. Emphasis, therefore, has been placed on a clean and open system structure, powerful system-building tools, and inter-project code sharing and reuse.

ControlShell is unique in offering:

- Functional object hierarchies, for *both* data-flow modules and action routines.
- Integrated state (strategic) and data-flow (servo-level) programming.
- Objects that retain their identities with an object name service, resulting in unlimited connectivity.
- Fine-grain blocks, executed in a single task context and ordered for minimal delay.
- Sophisticated operating mode (configuration) management.

ControlShell has recently been released as a commercial product. It has already found considerable application in universities, government, and industry. For instance, it has been embraced as the basis for most of the new robotics and the space-flight test-bed development at Jet Propulsion Laboratories, and selected as the controlling architecture for developing waste remediation systems by Battelle Pacific Northwest Laboratories and Oak Ridge National Laboratories.

Acknowledgments Portions of this work were supported under ARPA and NASA contracts. The authors wish to thank Dr. R. H. Cannon, Jr. of Stanford for his guidance and leadership. The authors would also like to thank the many developers at Stanford, NASA and other sites who have contributed help, suggestions, and software components.

References

1 D. B. Stewart, D. E. Schmitz, and P. Khosla, "The Chimera II: Real-Time Operating System for Advanced Sensor-Based Robotic Applications," IEEE Transactions on Systems, Man, and Cybernetics, vol 22, no 6, pp1282-1295, December 1192.

2 Wind River Systems, Inc., 1351 Ocean Ave., Emeryville, CA 94608, VxWorks User's Manual, 1988-1993.

3 Integrated Systems, Inc., 2500 Mission College Boulevard, Santa Clara, CA 95054, ISI Product Literature, 1990-94.

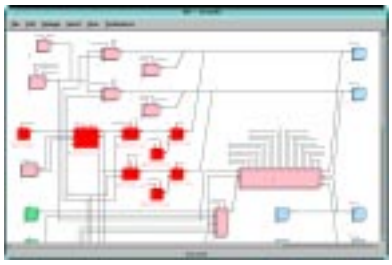
4 Ready Systems, Inc., VRTX User's Manual, 1994.

5 G. Pardo-Castellote and S. A. Schneider. The Network Data Delivery Service: Real-Time Data Connectivity for Distributed Control Applications. International Conference on Robotics and Automation, 1994.

6 Lumia, et. al., "NASREM Robot Control System Standard," Robotics and Computer Integrated Manufacturing, vol. 6, no 4, 1989

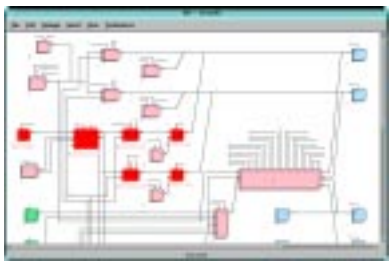
Independent Arm Trajectories

These two via-point trajectory components allow the planner to generate independent trajectories for each arm. A capture sequence is initiated using independent arm trajectories to approach the object coming down the conveyor belt.



Intercept Trajectories

As the arms approach the object on the conveyor belt, the system switches into an Intercept trajectory, where the actual measured positions of the conveyor belt (on the left) is used to generate the proper trajectories. Two fifth-order trajectory components plot an intercept path.



Tracking Trajectories

As the arms reach the object, the fifth-order trajectory components are disabled to allow direct tracking of the object. The sensed positions of the object are used directly servo the robot arms to capture the object.

Cooperative Motion

Finally, the system switches into the fully cooperative mode to move the object through the obstacle field under the direction of the path planner. A single via-point trajectory component is used to convert the path planner output into smooth trajectories for the object that then can be translated into smooth motions for the arm endpoints.

7 Harel, David, et. al., “StateMate: A Working Environment for the Development of Complex Reactive Systems,”, IEE Transactions on Software Engineering, V16, n4, April 1990

8 M. Leahey, “Universal Telerobotics Architecture Project”

9 R.~G. Simmons. “Structured Control for Autonomous Robots” IEEE Transactions on Robotics and Automation, 10(1), February 1994.

10 D. Simon, E. Coste-Maniere, Roger Pissard, “A Reactive Approach to Underwater-Vehicle Control: The Mixed ORCCAD/PIRAT Programming of the VORTEX Vehicle,” programme 4 - Robotique, Image Et Vision, Unite De Recherche - INRIA-SOPHIA ANTIPOLIS, Domaine de Voluceau Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France, November 1992.

11 The MathWorks, Inc., Cochituate Place, 24 Prime Park Way, Natick MA 01760, Product Literature, 1990-93.

12 S. Schneider, Experiments in the Dynamic and Strategic Control of Cooperating Manipulators. PhD thesis, Stanford University, Stanford, CA 94305, September 1989.

13 S. Schneider and R. H. Cannon, “Object Impedance Control For Cooperative Manipulation: Theory and Experimental Results,” IEEE Journal of Robotics and Automation, vol. 8, June 1992.

14 S. A. Schneider and R. H. Cannon, “Experimental Object-level Strategic Control with Cooperating Manipulators,” The International Journal of Robotics Research, vol. 12, pp. 338--350, August 1993.

15 S. A. Schneider, V. Chen, and G. Pardo, “ControlShell: a Real-Time Software Framework,” AIAA Conference on Intelligent Robots in Field, Factory, Service and Space, March 1994

16 G. Pardo-Castellote, T.-Y. Li, Y. Koga, R. H. C. Jr., J.-C. Latombe, and S. Schneider, “Experimental Integration of Planning in a Distributed Control System,” in Preprints of the Third International Symposium on Experimental Robotics, (Kyoto Japan), October 1993.

17 M.A. Ullman, Experiments in Autonomous Navigation and Control of Multi-Manipulator Free-Flying Space Robots. PhD Thesis, Stanford University, March 1993

18 G. Pardo-Castellote, S. Schneider, and R. Cannon, “Robotic Workcell Manufacturing without Scheduling or Fixturing”, IEEE Conference on Robotics and Automation, May 1995

19 B. Selic, G. Gullekson, and P. Ward, “Real-Time Object-Oriented Modeling”, Wiley and Sons, 1994

20 M. W. Gertz, D. B. Stewart, and P. K. Khosla, “A Software Architecture-Based Human-Machine Interface for Reconfigurable Sensor-Based Control Systems,” in Proc of 8th IEEE International Symposium on Intelligent Control, Chicago, Illinois, August 1993.